

# Event-B/SLP

Alexei Iliasov

Newcastle University, UK

**Abstract.** We show how the event-based notation offered by Event-B may be augmented by algorithmic modelling constructs without disrupting the refinement-based development process.

## 1 Introduction

One of the lessons of the DEPLOY project [5] is that the industrial application of formal modelling cannot fully succeed by employing just one notation, paradigm and methodology. In the case of Event-B [2], one of the language strong sides at the level of abstract design - a simple and versatile notation suitable for a wide range of abstractions - makes the language difficult to apply to concrete designs. Unstructured event-based models often become unwieldy long and verbose when design and implementation decisions are added.

In this paper we discuss a proposal to extend the event-based notation of Event-B with algorithmic constructs that permit an efficient specification of a large class of concrete designs.

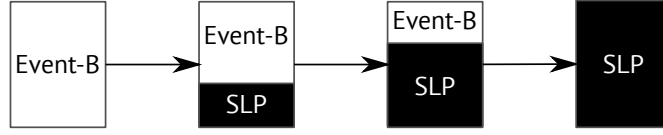
Our extension, language SLP (sequential composition, loop, parallel composition), is a compact formal modelling notation with strictly defined syntax and semantics. To stay on the same technological platform as Event-B, we define the language semantics as a list of FOL verification conditions. We adopt without changes the mathematical language of Event-B - the part of the notation used to define predicates and expressions. The languages also borrows the notation and the atomicity assumption of Event-B substitutions.

Rather than a replacement or a simple superposition of algorithmic and event styles we propose to have a seamless connection between Event-B and SLP where a high-level event specification is gradually transformed into an algorithmic specification with explicit concurrency and control flow (see Fig. 1).

The defining difference between SLP and Event-B is that the latter is data-driven while the former features explicit control flow for sequential computation and units of concurrency for concurrent computations. This requires a departure from a flat machine structure, apt for inductive reasoning but often onerous in practice for large models, to a hierarchical model with nested naming scopes delineating verification concerns.

## 2 Syntax

An SLP model is made of the following three main parts. The first, taken verbatim from Event-B, provides definitions of types, axiom, variables, invariants



**Fig. 1.** The SLP approach promotes a gradual transition from an event-based to an algorithmic specification.

and theorems. This part may also contain Event-B events in the case of a mixed Event-B/SLP model.

The second part is the definition of *environment* activities. In SLP, we take a view that actions performed by an environment must be explicitly defined as such. This is not just a syntactic notion - SLP offers differing refinement rules (not discussed in this paper) for environment and system activities.

The final part is the definition of the behaviour of a modelled system. It takes the form of a list of so-called *process* definitions - concurrent units of system behaviour. The body of a process is defined by a succession of *atomic state updates* (substitutions, in the Event-B terminology) connected by the typical algorithmic control structures - sequential composition, *if* and *loop*. A process body runs in an infinite loop until it explicitly executes a termination command.

The processes of a system and environment activities execute concurrently. They interact by reading and writing shared (global) variables. A system process may also define its private (local) variables to deal with computations that do not need to be exposed to environment or other system processes. For a given process, the *universe* of the process is the set of all other processes and all the environments.

The following is the top-level structure of an SLP specification:

$$\begin{aligned}
 slp &:= \langle invdef \rangle^* \\
 &\quad \langle environment \rangle^* \\
 &\quad \langle process \rangle^+ ; \\
 \langle invdef \rangle &:= (\text{invariant} \mid \text{theorem}) \langle label \rangle : \langle predicate \rangle ;
 \end{aligned}$$

To simplify the presentation, we omit the declaration of constants and sets while variable declarations are deduced from invariants.<sup>1</sup> All the variables defined at the global level are seen by system and environment processes. These should be the variables used to model input/output between the environment and system components. Like in Event-B, we split invariant conditions to label and partition invariant preservation conditions.

An environment is a labelled pair of a rely and guarantee predicates. Like invariants, rely and guarantees are labelled.

<sup>1</sup> Note that this is our preferred *concrete* syntax. The abstract syntax for these elements is exactly that of Event-B

$$\begin{aligned}
\langle environment \rangle &:= \text{environment } \langle label \rangle \langle reldef \rangle^* \langle gardef \rangle^* \text{end}; \\
\langle reldef \rangle &:= \text{rely } \langle label \rangle : \langle predicate \rangle ; \\
\langle gardef \rangle &:= \text{guar } \langle label \rangle : \langle predicate \rangle ;
\end{aligned}$$

The following is an example of an environment describing the behaviour of a temperature sensor. The environment may update value of  $t$  (current temperature) by changing it in some small increments defined by constant  $\Delta$ . A rely predicate is omitted and assumed to be  $\top$ .

```

environment temp_sensor
  guar guar1 is  $t' \in t - \Delta .. t + \Delta$ 
end

```

A system activity, called a process, follows the template of an environment but may also define local variables and concrete behaviour specification.

$$\langle process \rangle := \text{process } \langle label \rangle \langle reldef \rangle^* \langle gardef \rangle^* \langle invdef \rangle^* \langle block \rangle? \text{end};$$

Informally, the body of a process is the implementation that is shown to tolerate the interference defined by the process rely and satisfy the obligation of the process guarantee. In an extreme case of a solipsistic process there may be no rely and guarantee predicates so that the process has no specific obligations to its universe. Such a process specifies a sequential algorithm that runs till completion without any interaction.

Continuing the theme of the sensor example, with the syntax discussed, we can already define a small but meaningful specification. The temperature sensor  $t$  belongs to the environment while the system controls the heater modelled by variable *heater*:

```

invariant temp :  $t \in \mathbb{Z}$ 
invariant heater :  $h \in \text{BOOL}$ 
environment temp_sensor :
  guar guar1 is  $t' \in t - \Delta .. t + \Delta$ 
end
process heater_control
  rely rel1 :  $t \in \text{SAFE\_TEMP}$ 
  guar guar1 :  $t > \text{TEMP\_HIGH} \wedge h = \text{TRUE} \Rightarrow h' = \text{FALSE}$ 
  guar guar2 :  $t < \text{TEMP\_LOW} \wedge h = \text{FALSE} \Rightarrow h' = \text{TRUE}$ 
end

```

There may be any number of **environment** and **process** parts. One may, for instance, add an alarm process to detect an abnormal temperature range.

```

invariant alarm :  $alarm \in \text{BOOL}$ 
process alarm_control
  guar guar1 :  $alarm' = \text{bool}(t \notin \text{SAFE\_TEMP})$ 
end

```

Note that the rely of `heater_control` is not always satisfied by the sensor behaviour. A system process is temporarily disabled if its rely is broken by an environment. A process, however, may not violate the rely of another process or an environment.

The body of a process describes how the activity defined by its guarantee predicate is realised. The following operators are used to build the body of a process:

```

⟨block⟩      := ⟨action⟩ ; ⟨block⟩ ;
⟨action⟩     := ⟨statement⟩ atomic? ⟨refines⟩? ⟨with⟩?
⟨statement⟩ := ⟨substitution⟩ | ⟨if⟩ | ⟨loop⟩ | ⟨begin_end⟩ | ⟨assert⟩ | stop ;
⟨if⟩        := if ⟨predicate⟩ then ⟨block⟩
              (elseif ⟨predicate⟩ then ⟨block⟩)*
              (else ⟨block⟩)? end ;
⟨loop⟩      := while ⟨predicate⟩
              ⟨invdef⟩*
              var ⟨expression⟩
              then ⟨block⟩ end ;
⟨begin_end⟩ := begin ⟨invdef⟩* ⟨block⟩ end ;
⟨assert⟩    := (assert (⟨label⟩ :)? ⟨predicate⟩)+

```

Most of the syntax is self explanatory. The **stop** statement terminates a process; **assert**  $p$  asserts the truth of  $p$ ; ⟨substitution⟩ and ⟨expression⟩ are Event-B substitution and expression elements (see Rodin Deliverable D7 [7] for concrete definitions). Block **begin\_end** defines the scope of visibility for local variables. Elements **atomic**, ⟨refines⟩ and ⟨with⟩ are used to define the refinement relationship between SLP models but are not discussed in this paper.

A trivial implementation of `heater_control` retells the implications in the process guarantee as an *if* statement:

```

process heater_control
  rely rel1 :  $t \in \text{SAFE\_TEMP}$ 
  guar guar1 :  $t > \text{TEMP\_HIGH} + \delta \wedge h = \text{TRUE} \Rightarrow h' = \text{FALSE}$ 
  guar guar2 :  $t < \text{TEMP\_LOW} - \delta \wedge h = \text{FALSE} \Rightarrow h' = \text{TRUE}$ 
  if  $t > \text{TEMP\_HIGH} + \delta \wedge h = \text{TRUE}$  then
    act1 :  $h' := \text{FALSE}$ 
  elseif  $t < \text{TEMP\_LOW} - \delta \wedge h = \text{FALSE}$  then
    act2 :  $h' := \text{TRUE}$ 
  end
end

```

## 2.1 Semantics

Similar to Event-B, the semantics of SLP is given as a list of verification conditions called proof obligations. We discuss only the consistency conditions showing

that the SLP part of an Even-B/SLP model does not violate invariants and introduce deadlocks and divergences. Informally, the purpose of consistency proof obligations is to establish the following three facts:

- when control is passed to a statement, the state update defined by the statement may take place;
- any statement does not take the system outside of the safety invariant bounds;
- a statement eventually terminates.

We begin by cataloguing the major syntactic elements of a specification. The following are coming from Event-B and are shared between Event-B and SLP models: constants  $c$ , carrier sets  $s$ , axioms  $P(c, s)$ , global variables  $v$  and invariant  $I(c, s, v)$ .

There are elements specific to SLP. Taking the viewpoint of a substitution  $S$  located somewhere in the body of a process, they are: the rely  $R(c, s, v, v')$  and guarantee  $G(c, s, v, v')$  of a current process; process variables  $u$  (must be distinct from  $v$ ); process invariant  $T(c, s, v, u)$ ; variables defined in enclosing **begin**... and **while**... blocks,  $\mathbf{w} = \{w_1, \dots, w_i\}$  (all distinct); **begin**... and **while**... block invariants  $B_i(c, s, v, u, w_1, \dots, w_i)$ ; assertion predicate  $A(c, s, v, u, \mathbf{w})$  expressed directly in a preceding **assert** or derived from other kind of a preceding statement; and, finally, the substitution itself -  $S(c, s, v, u, \mathbf{w}, v', u', \mathbf{w}')$ .

The following shorthand is used to identify syntactic element in the context of substitution  $S$ . Assume that  $S$  is contained inside  $i$  nested blocks **begin/while** that define some local variables  $u$  and  $\mathbf{w}$ . In the scope of  $S(\dots)$  the actual invariant is  $\mathcal{I}_i$ , as defined below. The invariant defines the state space  $\Omega_i$  on which the update defined by  $S$  takes the effect:  $\{z \mid S(z)\} \subseteq \Omega_i \times \Omega_i$ .

$$\mathcal{I}_i = \left( \begin{array}{l} P(c, s) \\ I(c, s, v) \\ T(c, s, v, u) \\ \bigwedge_{j \leq i} B_j(c, s, v, u, w_1, \dots, w_j) \end{array} \right) \quad \begin{array}{l} \mathcal{A} = A(c, s, v, u, \mathbf{w}) \\ \mathcal{S} = S(c, s, v, u, \mathbf{w}, v', u', \mathbf{w}') \\ \Omega_i = \{z \mid \mathcal{I}_i(z)\} \\ \Omega_i^\vee = \Omega_i \cup \{\sqrt{\phantom{x}}\} \end{array}$$

Extended state  $\Omega_i \cup \{\sqrt{\phantom{x}}\}$  adds a termination symbol  $\sqrt{\phantom{x}}$  from which no continuation is possible. Globally, the set of all names spaces forms a tree such that the state of an inner wholly contains the state of outer space:  $\Omega_0 \subseteq \Omega_1 \subseteq \dots \subseteq \Omega_n$  where  $\Omega_0$  is the state of a name space of containing just global variables and  $\Omega_n$  is the state of some current block within the body of a process.

To define verification conditions, we convert SLP statements into relations describing the connection between previous and next states. All the partial state update relations are treated as *guarded relations* (i.e., never applied outside of their domain) and loops are required to terminate to ensure *total correctness*. We write  $\parallel$  meaning  $[\mathcal{I}_i]$ ,  $\mid$  meaning  $[I]$ ,  $\mathbf{A}$  for  $[\mathcal{A}]$  and so on.

$$\begin{array}{ll}
\llbracket \dots \rrbracket_i & \in \Omega_i \leftrightarrow \Omega_i^\vee \\
\llbracket \text{stop} \rrbracket_i & := \Omega_i \times \{\sqrt{\phantom{x}}\} \\
\llbracket \text{assert } p \rrbracket_i & := \text{id}([p] \cap \Omega_i) \\
\llbracket a ; \text{assert } p ; b \rrbracket_i & := ([p] \triangleleft \llbracket b \rrbracket_i)^\diamond \\
\llbracket a ; b \rrbracket_i & := \llbracket b \rrbracket_i \circ \llbracket a \rrbracket_i \\
\llbracket s_1 \parallel \dots \parallel s_n \rrbracket_i & := \llbracket s_1 \rrbracket_i \cup \dots \cup \llbracket s_n \rrbracket_i \\
\llbracket u := E(v) \rrbracket_i & := \{u \mapsto E(v)\}^\diamond \\
\llbracket u \in E(v) \rrbracket_i & := \{u \mapsto u' \mid u' \in E(v)\}^\diamond \\
\llbracket u : E(v, v') \rrbracket_i & := \{u \mapsto u' \mid E(v, v')\}^\diamond \\
\begin{array}{l} \llbracket \text{if } c_0 \text{ then } b_0 \\ \text{elseif } c_1 \text{ then } b_1 \\ \dots \\ \text{elseif } c_k \text{ then } b_k \\ \text{else } b_e \text{ end} \end{array} \parallel_i & := ((c_0 \triangleleft b_0) \cup (c_1 \triangleleft b_1) \cup \dots \cup (c_k \triangleleft b_e))^\diamond \\
& \text{where } c_i = c_i \setminus (\bigcup_{j=0..i-1} c_j) \\
\begin{array}{l} \llbracket \text{while } c \\ \text{invariant } LI \\ \text{var } V \\ \text{then } b \text{ end} \end{array} \parallel_i & := \llbracket \text{assert } \neg c \wedge LI \wedge \text{trm}(C, LI, V, \llbracket b \rrbracket_{i+1}) \rrbracket_i \\
\begin{array}{l} \llbracket \text{begin} \\ \text{invariant } BI \\ b \end{array} \parallel_i & := ([BI] \triangleleft \llbracket b \rrbracket_{i+1} \cap (\Omega_i \times \Omega_i))^\diamond \\
\llbracket \text{end} \rrbracket_i & 
\end{array}$$

Operator  $r^\diamond$  extends a relation  $r \subseteq \Omega_i \times \Omega_i^\vee$  to a total relation  $r' \subseteq \Omega_i \leftrightarrow \Omega_i^\vee$  so that mappings not covered by  $r$  are taken from  $\text{id}(\Omega_i)$ :  $r^\diamond = \{x \mapsto y \mid x \mapsto y \in r \vee x \mapsto y \in \text{id}(\Omega_i) \setminus r\} = \text{id}(\Omega_i) \triangleleft r$ . Also, as a shorthand, for some predicate  $x \in \Omega_I \rightarrow \text{BOOL}$  we write  $[x]$  to mean a set of elements satisfying  $x$ :  $[x] = \{e \mid x(e)\}$ .

In the definition of a loop,  $V \in \Omega_i \rightarrow \mathbb{N}$  is a loop variant and  $\text{trm}$  is a termination condition expressing that the variant value is decreased by each loop iteration:  $\text{trm}(C, LI, V, b) \equiv \forall x, y \cdot x \in V[b[st]] \wedge y \in V[st] \Rightarrow x < y$ , where  $st \equiv I \cap [LI \wedge c]$ .

Note the two rules for sequential composition. The  $a ; b$  case defines the conventional sequence-to-relational-join rule. The preceding rule (of a higher precedence) makes the sequential composition 'forgetful' when an assertion is placed between two statements: the information about previous statements is dropped and the focus is placed on the last statement and the preceding assertion. One reason for this is that a chain of substitutions may lead to a large and intractable set of hypothesis preventing efficient automated proof and introduce an undesirable interdependency between substitutions where a change in one substitution could invalidate proofs done for successive substitutions. An assertion breaks such a chain making the proof context smaller. Another reason, specific to our technique of refining Event-B into SLP, is the use of assertions

to prove that the set of enabling states of a refined substitution does not grow larger in a refined model.

The following is a list of the more important proof obligation, given, for brevity, in a relational form.

*Well-definedness* SLP mirrors the Event-B approach of proving that each partial relation is well-guarded. In other words, we prove that a relation defined by statement  $a$  may be applied to a current state:  $(\Pi \cap A) \triangleleft \llbracket a \rrbracket \neq \emptyset$ .

*Feasibility of rely* The rely must not contradict an invariant:  $I \triangleleft R \subseteq I \times I$ .

*Closure of rely* Conditions involving rely invariably require tolerating any number of rely iterations. To simplify corresponding proof obligations we insist that a rely relation  $R$  is reflexively and transitively closed:  $\text{id}(\Omega_i) \subseteq R \wedge R \circ R \subseteq R$ .

*Invariant preservation* Invariant properties of model variables are assumed to hold before every substitution. It must be proven that all invariants known in the scope of a substitution are re-established by the substitution:  $\llbracket a \rrbracket [\Pi \cap A] \subseteq \Pi$ . Note that when statement  $a$  is located in the body of a loop  $\Pi$  also includes the loop invariant.

*Variant* A loop variant is based on the same principles as Event-B variant and is embedded into the rule converting a loop into a relational form.

*Establishing guarantee* A substitution executed by a process must agree with a process guarantee. Formally, any state update would be covered by a 'promise' expressed in the guarantee:  $(\Pi \cap A) \triangleleft \llbracket a \rrbracket \subseteq G$ .

*Establishing assertion* An asserted condition  $A_n$  must be implied by a previous assertion or a statement. We must take into the account the fact that between previous and current statements the universe might have changed its state. For this, the latest locally known state is 'blurred' by the rely condition of a process.

- if two assertions follow each other then the second must be contained in the first:  $(A \triangleleft R) [\Pi] \subseteq A_n$ ;
- otherwise, if an assertion is preceded by a substitution, the preceding substitution after-state must imply the assertion:  $(R \circ \llbracket a \rrbracket) [\Pi] \subseteq A_n$ ;
- otherwise, an assertion must be established by an invariant:  $R[\Pi] \subseteq A_n$ .

*Process compatibility* All non-environment processes must be compatible w.r.t. their rely/guarantee conditions:  $I \triangleleft G_A \subseteq R_B$ .

### 3 From Event-B to SLP

SLP is not a standalone formalism and is meant to complement the Event-B notation when one needs to obtain a detailed design expressed in terms of parallel processes and algorithmic constructs. Thus, there is always a stage when a pure Event-B specification undergoes a transformation into an Event-B/SLP specification.

One simple case of Event-B to Event-B/SLP refinement is introducing environments and processes operating on new variables. In a general case, the Event-B part is refined to make use of new variables so that there is an information flow between the two parts. Naturally, there are no specific proof obligations for this case: one only needs to discharge the consistency conditions.

A more interesting situation is the *replacement* of Event-B events with SLP constructs. Of all possibilities, we shall only consider the simplest one: refinement of a set of events by *new* (rather than existing) environments and processes.

*New environment (process)* A new SLP environment (process) may be defined to refine one or more abstract Event-B events; refined events disappear from a model. The relevant proof obligation is that a process guarantee is contained in the behaviour of refined events:  $(I \cap R) \triangleleft G \subseteq [e_1]_R \cap \dots \cap [e_n]_R$ .

*New concrete process* A sub-set of machine events may be refined into a process with a body. We focus on a simpler case when this is done in a single refinement step. Without loss of generality, we consider the case of refinement where substitutions of a process body coincide exactly with substitutions of refined events, in other words, a refinement that forms a process from events without any further behavioural or data refinement that may take place in following refinement steps.

Let  $E$  be the set of events of machine  $M$  describing the behaviour of a prospective process  $P$  and  $tr(M) \upharpoonright E$  be the machine traces limited to events  $E$ . Let  $tr(P)$  be a set of traces of a new process in terms where each trace element is the list of labels of parallel substitution parts. It is easy to define a mapping  $f$  from the alphabet of  $tr(P)$  to set  $E$  (it is not necessarily a one-to-one mapping but this does not pose problems). If one can prove that  $f(tr(P)) \subseteq tr(M) \upharpoonright E$  and, separately, that process  $P$  does not introduce new divergences then process  $P$  is declared to refine events  $E$ . We have previously shown how to convert a statement of the form  $f(tr(P)) \subseteq tr(M) \upharpoonright E$  into a list of FOL theorems [3,4].

### 4 Small Example

We illustrate the Event-B/SLP hybrid modelling by showing a simple case of Event-B to SLP refinement. The model computes the greatest common divisor (GCD) of two numbers. Function  $\text{gcd} \in \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  axiomatically satisfies the following properties:



$$\begin{aligned}\text{axm1} &: \forall a, b \cdot a, b \in \mathbb{N} \wedge a > b \Rightarrow \text{gcd}(a, b) = \text{gcd}(a - b, b) \\ \text{axm2} &: \forall a, b \cdot a, b \in \mathbb{N} \wedge b > a \Rightarrow \text{gcd}(a, b) = \text{gcd}(a, b - a) \\ \text{axm3} &: \forall a \cdot a \in \mathbb{N} \Rightarrow \text{gcd}(a, a) = a\end{aligned}$$

At an abstract level, one may use the constant function gcd to compute the result in one step:

```
machine gcd0
  variables r, x1, x2
  invariant r ∈ ℕ ∧ x1 ∈ ℕ ∧ x2 ∈ ℕ
  initialisation r := 0 || x1 := 0 || x2 := 0
  events
    gcd = begin r := gcd(x1 ↦ x2) end
end
```

Variables  $x1$  and  $x2$  serve as input values and  $r$  holds the result. The following is a typical Event-B refinement based on the unfolding of an atomic abstract step into a sequence of concrete computations.

```
refinement gcd1a
  refines gcd0
  variables r, x1, x2, y1, y2, pc
  invariant
    y1 ∈ ℕ ∧ y2 ∈ ℕ
    pc ∈ 1 .. 5
    pc = 2 ⇒ gcd(x1 ↦ x2) = gcd(y1 ↦ x2) ∧ y1 > 0 ∧ x2 > 0
    pc = 3 ⇒ gcd(x1 ↦ x2) = gcd(y1 ↦ y2) ∧ y1 > 0 ∧ y2 > 0
    pc = 4 ⇒ gcd(x1 ↦ x2) = gcd(y1 ↦ y2) ∧ y1 > 0 ∧ y2 > 0
  initialisation ... || y1 := 0 || y2 := 0 || pc := 1
  events
    copy1 = when pc = 1 then y1 := x1 || pc := 2 end
    copy2 = when pc = 2 then y2 := x2 || pc := 3 end
    sub1  = when y1 > y2 ∧ pc ∈ {3, 4} then y1 := y1 - y2 || pc := 4 end
    sub2  = when y2 > y1 ∧ pc ∈ {3, 4} then y2 := y2 - y1 || pc := 4 end
    gcd   = when y1 = y2 ∧ pc = 4 then r := y1 end
  end
```

Events `sub1` and `sub2` form the body of a loop. An auxiliary variable  $pc$  is used to simulate control flow; variables  $x1, x2, y1, y2$  are introduced to describe the concrete computation steps. Note how the after state of each event is encoded in model invariant. The repeating template  $v = C \Rightarrow \dots$  in invariants is an indicator that an event-based specification is used to simulate concrete control flow.

The SLP version of the same refinement step is given below. Here we have an explicit loop construct containing a two-branch *if* that makes for a more concise specification without the need to propagate state properties via an invariant.

```

refinement gcd1b
  refines gcd0
  variables  $r, x1, x2, y1, y2$ 
  invariant  $y1 \in \mathbb{N} \wedge y2 \in \mathbb{N}$ 
  initialisation ...  $\parallel y1 : \in \mathbb{N} \parallel y2 : \in \mathbb{N}$ 
  process main
     $y1 := x1 \parallel y2 := x2$  ;
    while  $y1 \neq y2$  then
      invariant  $gcd(x1 \mapsto x2) = gcd(y1 \mapsto y2) \wedge y1 > 0 \wedge y2 > 0$ 
      if  $y1 > y2$  then  $y1 := y1 - y2$ 
      elseif  $y2 > y1$  then  $y2 := y2 - y1$  end
    end ;
     $r := y1$ 
  end
end

```

Essential to the proof of refinement is the last case of sequential composition where control is passed from a loop to an assignment saving the final result. The relational interpretation of the loop asserts the loop invariant and the negation of the loop condition which immediately give that  $r = y1 = gcd(x1 \mapsto x2)$ .

## 5 Conclusion

The implementation language of B-Method, B0 [1] is one of the inspirations for this work. There are, however, important differences in both aims and techniques employed: B0 allows a modeller to write more detailed bodies of abstract operations using the concepts from programming languages. In contrast, in SLP, the main development technique is an aggregation of several abstract events into a body of a process. This means that a data-driven design of Event-B may be refined into an algorithmic design whereas in B0 it would have to remain data-driven at the top level. Equally important is an explicit treatment of concurrency that becomes more and more relevant topic in embedded systems design. We use rely/guarantee [6] approach to model cooperation of concurrent processes via shared variables.

Event-B is rather obviously lacking in means of control flow specification. One solution is the integration of two narrowly specialised two notation, i.e., CSP||B that combines B and CSP [9]. Another is extension of the basic notation with means to explicitly define control flow, i.e., the Flow plug-in for Rodin [3]. In this paper we followed a different direction with a premise that a deficiency of a notation in a certain area is best rectified by coming up with a new notation.

This leads us to the following crucial point: to make Event-B applicable in any given problem domain it may be necessary to (1) design a specialised concrete syntax exposing Event-B method in a way tailored to the problem domain (for example, a graphical notation like the one offered by UML-B [8]) and (2) devise a specialised notation and refinement rules for concrete designs, like the one shown in this paper. The use of Event-B for an abstract design puts a development on a solid and well-studied platform. But concrete designs incorporating implementation decision must offer the concepts, terminology and

structuring principles already employed and recognised in the target problem domain. In this sense, the language defined in this paper is merely a technological demonstration that such a direction is viable.

## References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. J.-R. Abrial. *Modelling in Event-B*. Cambridge University Press, 2010.
3. A. Iliasov. Use case scenarios as verification conditions: Event-B/Flow approach. In *Proceedings of 3rd International Workshop on Software Engineering for Resilient Systems*, Septembre 2011.
4. A. Iliasov. Augmenting formal development with use case reasoning. In *Ada Europe 2012*, June 2012.
5. Industrial deployment of system engineering methods providing high dependability and productivity (DEPLOY). IST FP7 project, online at <http://www.deploy-project.eu/>.
6. C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.
7. Rigorous Open Development Environment for Complex Systems (RODIN). Deliverable D7, Event B Language, online at <http://rodin.cs.ncl.ac.uk/>.
8. C. Snook and M. Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol*, pages 92–122, 2006.
9. H. Treharne, S. Schneider, and M. Bramble. Composing Specifications Using Communication. In *Proceedings of ZB 2003: Formal Specification and Development in Z and B, Lecture Notes in Computer Science*, Vol.2651, Springer, Turku, Finland, June 2003.